HARDWARE IMPLEMENTATION AND EVALUATION OF THE SPANDEX CACHE COHERENCE PROTOCOL

BY

ZERAN ZHU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Sarita Adve

Abstract

Emerging heterogeneous hardware systems and applications that have shared data between multiple CPU cores and computation accelerators bring the need for efficient and flexible cache coherence support. Since different devices like CPUs, GPUs and accelerators have diverse memory demands and different data-sharing patterns, Spandex was proposed to efficiently integrate devices with different cache coherence protocols. The flexibility, simplicity and scalability of Spandex make it suitable for maintaining cache coherence in complicated SoCs. In addition, the introduction of Flexible Coherence Specialization (FCS) in Spandex further improves the granularity of flexibility from device granularity to address and request granularity. However, even though benchmark evaluations of Spandex in simulation have shown significant benefits, it has not been proven that Spandex will perform as well in real hardware, due to the lack of real-world RTL implementation of the protocol itself. In this work, we implement the Spandex cache coherence protocol in real hardware and evaluate its performance on real system-on-chip architectures running on FPGAs. By implementing and integrating Spandex on a real-world FPGA SoC, we advance the Spandex protocol from software simulation to a real hardware implementation. We prove its efficiency and flexibility, which are the key benefits of Spandex already proven in simulation, but on the next level down to the hardware. On the Xilinx VCU118 FPGA evaluation platform, we evaluated Spandex by running hardware-accelerated micro-benchmarks on heterogeneous SoCs with the Spandex protocol compared to the MESI protocol. On these micro-benchmarks, we see a performance improvement of up to 1.77X, and also up to 3.55X and 5.30X network traffic improvement in terms of flit count and flip-hop count respectively. We also propose the Spandex RISC-V instruction set extension, as a new interface for Spandex-aware and FCS-aware applications to convey flexible coherence performance information down to the hardware. We also provide a configuration register based interface for easily managing coherence specializations for fixed-function accelerators that are not capable of executing dynamic code. The RTL implementation, along with the accompanying RISC-V ISA support, greatly reduces the obstacles to the adoption of Spandex in the research community, and allows more system designers to consider Spandex as their coherence solution to further boost performance.

Acknowledgments

This project would not have been possible without the support of many people.

I would like to thank Professor Sarita Adve for her inspiring advising and guidance that led me through my entire master's research project. Without her support, I could not have developed the entire cache system from the ground up and integrated Spandex into ESP.

I also would like to thank one of my collaborators, Dr. Paolo Mantovani, the main architect of ESP, for his help and support with this project. He is very nice to people, patient with questions, and open to collaborations. The kindness and collaborative attitudes I learned from him form a very important aspect of my graduate study.

I also would like to thank Robert Jin for spending days and nights with me together working on parts of this project. Robert's contributions to this work account substantially for the success of this project.

Many more people provided help and support during this project, including John Alsop, Luca Carloni, Wesley Darwin, Davide Giri, Vignesh Suresh, and Joseph Zuckerman.

This work is supported in part by the National Science Foundation under grant CCF 16-19245, by DARPA through the Domain-Specific System on Chip (DSSoC) program, a Google Faculty Research award, and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Table of Contents

List of Tables

List of Figures

Listings

List of Abbreviations

AMO	Atomic Memory Operation.
AQ	Acquire.
AXI	Advanced eXtensible Interface.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
DRF	Data-Race-Free.
ESP	Embedded Scalable Platforms.
FCS	Fine-grained Coherence Specialization.
FF	Flip-Flops.
FFT	Fast Fourier Transform.
FPGA	Field-Programmable Gate Array.
FSM	Finite-State-Machine.
FWD	Forward.
GNU	GNU is Not Unix.
GCC	GNU Compiler Collection.
GF-12	GLOBALFOUNDRIES 12 nm technology.
GPU	Graphics Processing Unit.
GUI	Graphical User Interface.
HLS	High-Level Synthesis.
ILP	Instruction-Level Parallelism.
IO	Input/Output.
IP	Intellectual Property. Also, Internet Protocol (in TCP/IP).
IPC	Instructions per Cycle.
ISA	Instruction Set Architecture.
L1	Level 1 (cache).

L2	Level 2 (cache).
LLC	Last-Level Cache.
MESI	Modified-Exclusive-Shared-Invalid (cache coherence protocol).
MSHR	Miss Status Holding Register.
NIC	Network Interface Card.
NoC	Network-on-Chip.
PSO	Partial Store Ordering.
REQ	Request.
RISC	Reduced Instruction Set Computer.
RISC-V	A Type of RISC Architecture.
RL	Release.
ROM	Read-Only Memory.
RSP	Response.
RTL	Register Transfer Level.
RVWMO	RISC-V Weak Memory Ordering.
SC	Sequential Consistency.
SLD	System-Level Design group.
SMP	(Linux) Simultaneous Multi-Processing.
SPARC	Scalable Processor ARChitecture.
SRAM	Static Random-Access Memory.
SoC	System-on-chip.
TCP	Transmission Control Protocol.
TSO	Total Store Ordering.
TU	Translation Unit.
UART	Universal Asynchronous Receiver-Transmitter.

Chapter 1

Introduction

1.1 Motivation

Due to the end of Dennard scaling and Moore's law [Taylor, 2013], specialization has become the focus of computer architecture design. More and more applications suffer from less-efficient CPUs or even GPUs, and the need for more specialized hardware accelerators is ever increasing. Thus, SoC designers tend to use more processing cores and more dedicated hardware accelerators. Such SoCs with multiple processor cores and various kinds of other processing units like GPUs and dedicated hardware accelerators are called "heterogeneous systems".

In a heterogeneous system, how to maintain data sharing is a hard problem. Traditionally, when a task is to be offloaded to a co-processor, the programmer would have to explicitly copy the shared input data onto the co-processor memory, and again explicitly copy back the output when it is done. This is very inefficient and inflexible because programmers need to conservatively pay unnecessary movements of data for programs with fine-grained synchronization and irregular memory access patterns. This is costly and wasteful, posing more threats to the memory bottleneck which is already slowing things down.

Shared-memory space is, on the other hand, another option. However, allowing different devices to access the same memory address space while maintaining coherence and without bringing complexity is a hard problem. Different processors and accelerators have different memory needs. Thus, they use different cache coherence protocols, such as MESI, GPU or DeNovo, [Alsop et al., 2018] discussed in detail later. Integrating different cache coherence protocols across different devices can be solved by using Spandex [Alsop et al., 2018].

Spandex aims to serve the needs of an efficient cache coherence protocol for large-scale heterogeneous systems. Spandex is essentially a flexible interface that provides seamless integration of multiple types of coherence protocols serving different private caches. With its design methodology for designing large-scale SoCs, Spandex aims to solve the dilemma of choosing the correct cache coherence protocol for large-scale heterogeneous systems. Traditionally, CPUs rely on a pure hardware coherence protocol such as MESI, whereas accelerators with private caches are plugged into an inefficient translation interface that translates the accelerator's memory accesses to make them compliant with the CPU's MESI. This type of translation wastes on-chip area, power, and a lot of design and verification effort of

the SoC designer and meanwhile hurts the performance of the accelerators by forcing them to use the complex and inefficient MESI style coherence messages. Spandex, on the other hand, provides a flexible and scalable communication interface between the CPUs and other types of accelerators, allowing them to use any type of protocol they want without harming data coherence or computation performance.

Moving beyond device-granularity coherence flexibility in Spandex, Fine-grained Coherence Specialization [Alsop et al., 2021], or FCS, provides even more flexibility at the address and request granularity. It also introduces new request types such as write-through forwarding and owner-prediction, in order to further exploit locality and improve communication efficiency.

Though Spandex has existed in simulators for years and has been exposed to some research work, it has not, to date, come down to the level of real hardware evaluations. This is a huge barrier to its adoption by many more SoC designers. In order to prove the flexibility and scalability that Spandex provides as a cache coherence interface, and in order to demonstrate the design methodology and performance benefits of using Spandex, a hardware implementation of Spandex is required.

1.2 Contributions

1.2.1 Protocol Implementation

We implement the Spandex cache coherence protocol in real hardware running on ESP [Carloni, 2016], a scalable SoC generator provided by the SLD group at Columbia University. ESP already provides modularized, network-on-chip (NoC) based integration of many CPUs and many accelerators with the support for MESI cache coherence protocol, so developing Spandex on it only requires a modest amount of architectural changes, which will be discussed in detail later. Most of the required work is within the hardware development of the caches themselves, and the integration process with ESP takes only minimal changes to the core-cache interface as well as the NoC interface. This design effort aligns with the Spandex flexibility methodology: any system that wants to use Spandex can easily adopt it without significant changes to their existing hardware platforms.

During this project, the author led and did most of the work in Spandex RTL design and implementation, especially the ESP integration, all of Spandex LLC, all of MESI TU, and the majority of Spandex FCS private cache development, while work done by others concentrated on other aspects such as RISC-V ISA extension implementation by Robert Jin, and RISC-V fence instruction support by Vignesh Suresh. Robert Jin also contributed to the extensive testing of the hardware implementation.

1.2.2 Processor Interface (RISC-V ISA Extension)

RISC-V is the target architecture we are using in this project. We provide a complete suite of Spandex FCS optimization instructions that comply with the existing RISC-V ISA Specification. We leverage the custom instruction opcodes in the RISC-V Specification to design a standardized interface for Spandex-aware software to communicate with the underlying Spandex cache about FCS information including flexible types of Spandex load and store requests, as well as owner prediction information. The standardized interface enables the development of portable software that can run on other future hardware platforms that support Spandex. The proposed new RISC-V instructions that support FCS information are implemented in the Ariane (now called CVA6) processor [Zaruba and Benini, 2019] by a small number of modifications to the decoder, the cache load and store unit, as well as the output channels in the AXI bus interface. Importantly, the changes to the Ariane core and its underlying bus interconnect only add additional output signals to the Spandex private cache interface, and do not add any additional input to the processor core, which keeps the re-verification effort of the processor small.

1.2.3 Accelerator Interface

We design and implement a full configuration interface for non-programmable accelerators (fixed hardware function accelerators that do not execute dynamic code) to use Spandex optimizations. This configuration interface allows the Spandex FCS information to be runtime configurable to the accelerator from the CPU, providing the software with a flexible way to dynamically configure and fine-tune the coherence specialization in memory load and store requests generated by the accelerator's private cache.

1.2.4 Evaluation

We evaluate the Spandex performance on real system-on-chip architectures running on FPGAs. Comparing Spandex performance with other coherence strategies in ESP including MESI, we demonstrate the benefits of using Spandex as the underlying cache hierarchy for heterogeneous systems.

1.3 Impact

By implementing and integrating Spandex on the ESP hardware SoC, we advance the Spandex protocol from software simulation to a real hardware implementation and demonstrate its performance benefits compared to other coherence strategies in ESP including the MESI coherence protocol. The formalized and standardized RISC-V ISA extension and the run-time configurable accelerator configuration interface pave the way for smart software and compilers that can benefit from using Spandex. Furthermore, this work greatly reduces the resistance to the adoption of Spandex

by the research community, and enables more system designers to consider Spandex as their coherence solution to further boost performance. This work also opens the door for more developers and researchers to join the Spandex community and experiment with more interesting projects together.

Chapter 2

Background

2.1 Platform Selection

In order to implement Spandex in hardware, there are many open-source system-on-chip (SoC) design platforms available, such as OpenPiton [Balkind et al., 2016] and Embedded Scalable Platforms (ESP) [Giri et al., 2018]. Both platforms are scalable, modular, and flexible. They are both equipped with a MESI style cache coherence protocol as a starting point. Due to these common properties, they are both great candidate platforms on which to implement Spandex. However, at the start of this project, due to ESP's support for heterogeneous computing with many cores and many accelerators, we decided to use ESP as the basis of Spandex hardware development.

2.2 ESP Overview

ESP [Giri et al., 2018] is an open-source SoC design platform provided by the System-Level Design (SLD) group at Columbia University. ESP provides an efficient and scalable way of integrating multiple types of processors and accelerators in an SoC through a network-on-chip architecture, while supporting multiple ways of maintaining data sharing between the processors and accelerators.

Figure 2.1, shows a typical ESP modular SoC is composed of several types of compute tiles. This is a screenshot of the ESP SoC graphical configuration tool, which is the first step towards prototyping an SoC using ESP. Each tile includes the main functional component, such as compute units like CPUs (red) or accelerators (cyan), services like cache integration and configuration registers, and network-on-chip services such as router nodes that handle the network packet transmitting and receiving. There are two special kinds of tiles in ESP, namely, the memory tiles (blue) and the IO tiles (yellow). Each memory tile contains an optional LLC and the memory controller that communicates with the system main memory. Each IO tile contains miscellaneous IO components such as the Ethernet interface controller, UART interface controller and the CPU Boot Read-Only Memory (Boot ROM). All the tiles on ESP are connected by a 2-dimensional mesh network. The NoC has multiple planes such as Request, Forward and Response in order to prevent deadlocks from happening.

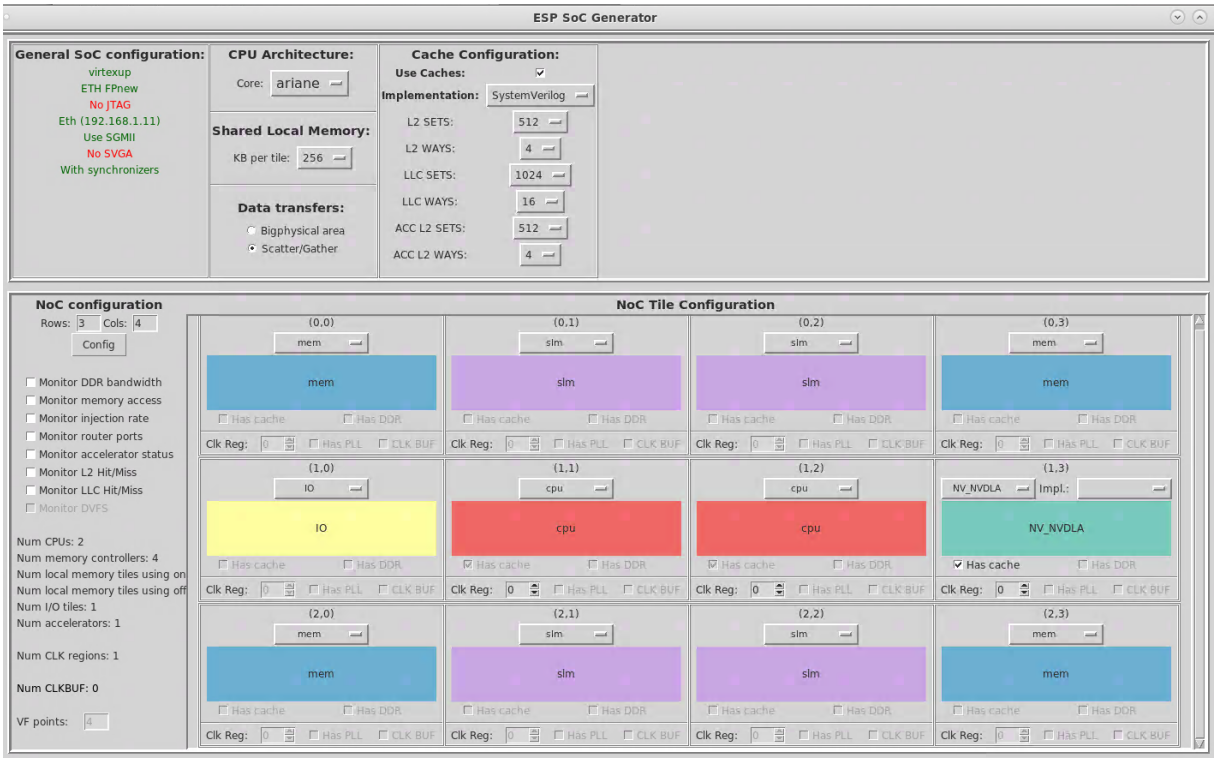

Figure 2.1: ESP Configuration Tool

In ESP, the following data sharing strategies are supported:

- Full MESI style cache coherence with in-core level 1 (L1), in-tile level 2 (L2) as well as last-level cache (LLC)
- LLC level direct memory access (DMA). In this mode, processors and accelerators access LLC directly, bypassing their private caches. To maintain cache coherence with the rest of the system, either software needs to flush relevant data of other private caches (LLC-coherent mode) prior to the beginning of a DMA transfer, typically before an accelerator starts running, or LLC needs to send downgrade messages to relevant private caches (recall-coherent mode).
- Non-coherent DMA. In this mode, processors and accelerators directly access the system main memory, bypassing even the LLC. The entire cache hierarchy, including all private caches and the LLC, needs to be flushed by the software, at the same point as in LLC-coherent mode with software flush.

Even though ESP already provides a useful NoC architecture and easy integration of multiple compute units and caches, integrating Spandex into ESP still requires the following work. First, since ESP only uses the MESI protocol whereas Spandex uses other protocols like graphical processing unit (GPU) style protocol and the DeNovo protocol [Choi et al., 2011], the current ESP network needs to be slightly modified for Spandex because of the encoding of different message types communicated via the NoC, the need for a special word-granularity bitmask with each

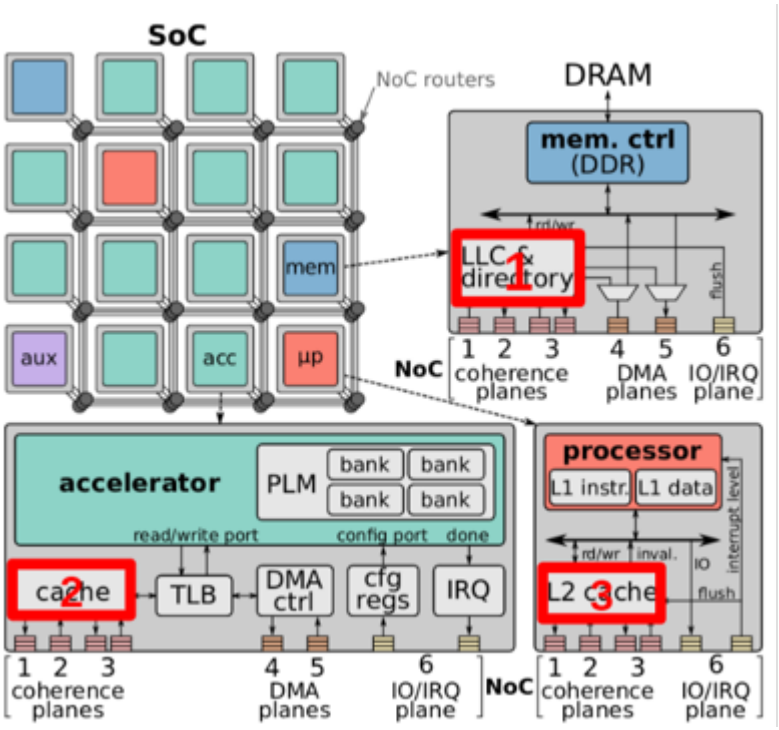

Figure 2.2: ESP Architecture

request, as well as special requirements for Spandex direct communication between private caches. Second, Spandex caches rely on specialized instructions to perform synchronization operations such as self-invalidations and write-buffer flushes when the program enters different phases. Such synchronizations are annotated as acquire (AQ) and release (RL) in the RISC-V architecture, which is the CPU architecture that will be used for this project. For each load or store instruction that requires synchronization, the transactions reaching the Spandex private cache interface must be annotated with the AQ and RL information so that the correct synchronization on the Spandex cache can be performed. This requires modifications to Ariane [Zaruba and Benini, 2019], the RISC-V intellectual property (IP) core used in the ESP project as well as this work. Third, the MESI private cache and the MESI LLC need to be completely replaced with the Spandex counterpart. To achieve this, the Spandex L2 cache and the Spandex LLC need to be implemented in hardware, which is part of the main goal of this work. For RISC-V Ariane, there is also an L1 cache which is embedded in the CPU. It does not need to be changed because communication between L1 and L2 such as read, write-through and invalidation messages is local to the tile, not requiring interaction with the NoC in Spandex coherence messages. Figure 2.2 (annotated) from the ESP paper [Giri et al., 2018] represents the ESP architecture. The components circled by red rectangles represent the caches that need to be changed to the Spandex version.

To summarize, the work left to integrate Spandex into ESP can be easily categorized into the following separate steps:

- Architectural changes to the ESP NoC
- Instruction set architecture (ISA) level changes to the RISC-V Ariane processor core in ESP
- The actual protocol development which involves writing the RTL code or generating RTL code from high-level behavioral code like SystemC.

2.3 RISC-V Ariane Overview

The RISC-V Ariane CPU IP core [Zaruba and Benini, 2019] (now called CVA6) is a 6-stage, single issue, in-order CPU which implements the 64-bit RISC-V instruction set. It fully implements I, M, A and C extensions as specified in RISC-V ISA Specification Volume I: User-Level ISA V 2.3 [RISC-V, 2019a]. It also implements three privilege levels M, S, U [RISC-V, 2019b] to fully support a Unix-like operating system.

At the time of this project, the processor IP cores available on ESP include LEON3 (SPARC V8 architecture) and Ariane (RISC-V architecture). At the early stages of ESP, due to the lack of ability to use caches for RISC-V Ariane, we initially developed the MESI TU on LEON3 and successfully booted Linux in SMP mode. After the initial success, due to its wide acceptance in many related works and research groups, RISC-V became a reasonable target ISA for designing the Spandex ISA extension. After the ISA extension for RISC-V architecture was formalized, the Spandex RTL development and testing effort has been focusing on the RISC-V Ariane processor.

2.4 Cache Coherence Protocols

Different devices have different memory needs and data access patterns. Thus, there are several different cache coherence protocols designed to suit the needs for different types of devices and workloads, among which, MESI (and its variants), GPU coherence and DeNovo are three commonly used cache coherence protocols.

2.4.1 MESI Coherence Protocol

MESI, or the Illinois Protocol is a very widely adopted cache coherence protocol. In MESI, there are four stable states possible for a given cache line from the perspective of a private cache.

- Modified. This state marks the ownership of a dirty cache line by a single private cache. In all other private caches and in main memory there is no valid copy of this cache line.
- Exclusive. This state marks the exclusive presence of a clean cache line in a single private cache. Other private caches do not keep the cache line and must send requests to read or write the content in this line.

- Shared. This state marks the presence of a clean cache line in more than one private caches.
- Invalid. This state indicates that the line is not valid in the private cache under consideration.

There are different ways in which MESI coherence actions can be done. For a snoop-based design, each write from a private cache requesting ownership must be broadcast to every other private cache, and other MESI private caches need to snoop write operations on the bus in order to invalidate the cache line being written to. For a directory-based design, there is a bookkeeping directory that keeps track of all current owners and sharers. When coherence actions are needed, the directory is responsible for sending out coherence messages such as ownership request/downgrade and invalidations to appropriate private caches.

2.4.2 GPU Coherence Protocol

Compared to MESI, GPU coherence protocol is relatively simple. It leverages the Data-Race-Free (DRF) memory consistency model and self-invalidates stale data in a core's own private cache upon software acquire synchronization points. Another difference between GPU coherence and MESI is that GPU coherence protocol does not obtain ownership on write. Instead, a GPU cache uses write-through on the next release synchronization to directly write up-to-date data to the next-level cache. Due to its simplicity of self-invalidation and write-through without ownership, GPU coherence is suitable for applications and devices optimized for high throughput.

2.4.3 DeNovo Coherence Protocol

DeNovo serves as an intermediate protocol between MESI and GPU so that it has the benefits of both MESI and GPU coherence. For writes, DeNovo protocol uses word-granularity ownership as opposed to line-granularity in MESI. For reads, DeNovo protocol uses self-invalidation just like in GPU coherence. For DRF programs, DeNovo is able to provide both low-latency benefits by obtaining ownership (exploiting data re-use across synchronization points) on write and high throughput benefits by utilizing self-invalidated reads. Table 2.1 (reprinted from the Spandex paper [Alsop et al., 2018]) summarizes the differences and similarities across MESI, GPU coherence and DeNovo.

2.5 Limitation of Existing Systems

Traditionally, in multi-core processors like the state-of-the-art x86 CPUs, the MESI style cache coherence protocols have served their purposes well [Alsop et al., 2018]. However, as many large-scale systems start to emerge, the MESI protocol has begun to evince some drawbacks.

First, MESI assumes writer-initiated invalidations. Whether relying on a snoop-bus based design or a directory-based design, a writer who updates any part of a cache line is responsible for propagating that information to every

Table 2.1: Coherence Strategy Classification

Coherence Strategy	Stale invalidation	Write propagation	Granularity
MESI	writer-invalidation	ownership	line
GPU Coherence	self-invalidation	write-through	loads: line stores: word
DeNovo	self-invalidation	ownership	loads: flexible stores: word

private cache in the system that could otherwise contain stale data. As the system grows with more numerous processing units, such writer-initiated invalidation messages could cause a lot of communication overhead that wastes not only time but also energy.

Second, MESI tracks all states at line granularity. This property of MESI causes unnecessary false sharing overhead when two processing units access different parts of a single line.

Third, the MESI coherence protocol intends to serve inherently racy accesses. Thus, many transient states are incurred to ensure that racy accesses are serviced properly according to the defined memory model of the system. These transient states on both the private caches and the last-level cache (LLC) impede performance and add complexity to the protocol.

Finally, MESI does not suit the needs of all kinds of processing units and programs. Some accelerators whose memory access patterns are simple and predictable can benefit from a cache coherence protocol with more flexibility and less complexity, rather than MESI. Also, for programs assuming the data-race-free (DRF) [Adve and Hill, 1990] programming model, the complexity of MESI is unnecessary because there is no data race between synchronizations.

GPU coherence, on the other hand, has its own limitations. For example, the self-invalidation and forced write-through flushing reduce the chance for exploiting data re-use across synchronization points. Also, all synchronization must be performed at the last level cache, which is very inefficient and costly.

With different characteristics of different cache coherence protocols and different memory demands of different specialized devices, it is hard to integrate an SoC with many processors and accelerators using different cache coherence protocols.

2.6 Spandex

Spandex is a flexible interface that provides support for all of MESI, GPU and DeNovo. It allows device-level coherence specialization with each device capable of running its own cache coherence protocol. In an SoC, Spandex fits different needs for diverse memory demands, just as a re-sizable glove fits different hands, as shown in figure 2.3, taken from the lightning talk of the Spandex paper [Alsop et al., 2018] in ISCA'18. The Spandex directory or LLC

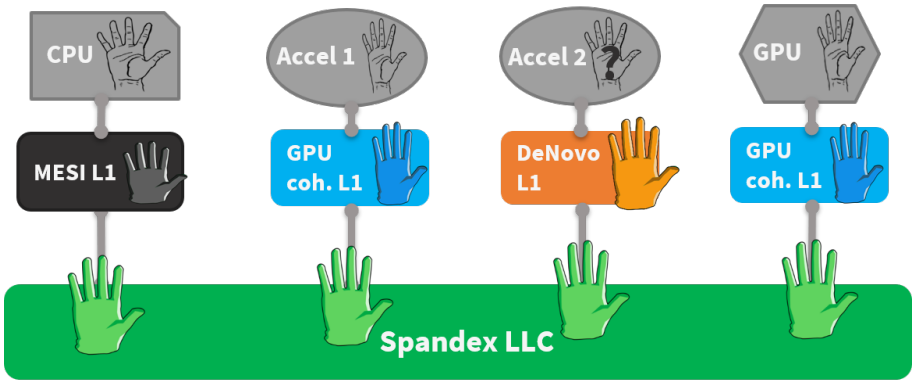

Figure 2.3: Spandex Working Like a Glove

integrates functionalities across different coherence protocols such as word-granularity owner tracking, write-through handling, line-granularity sharer tracking, etc. Among the three cache coherence protocols, DeNovo and GPU naturally work in Spandex with little overhead. MESI style coherence messages and states, together with the existing messages and states of the DeNovo and GPU protocols, form the set of all Spandex messages and states, which are listed in the following subsections.

2.6.1 LLC Stable States

In Spandex directory, only Shared state is tracked at line granularity. The following stable states are possible in the Spandex LLC.

1. Invalid. The word is not valid.
2. Owned (Registered). A word in the line is owned. Other words must be Owned or Valid.
3. Valid. The word is valid.
4. Shared. A cache line has been read by one or more writer-invalidated readers who are in the sharer list.

2.6.2 Device Requests

Spandex supports the following device requests in table 2.2 as listed in the Spandex paper [Alsop et al., 2018].

Table 2.2: Spandex Device Request Types

Device Type	Device Request	Spandex Request	Granularity
GPU Coherence	Read	ReqV	line
	Write	ReqWT	word
	RMW	ReqWT+data	word
DeNovo	Read	ReqV	flexible*
	Write	ReqO	word
	RMW	ReqO+data	word
	Owned Repl	ReqWB	word
MESI	Read	ReqS	line
	Write	ReqO+data	line
	RMW	ReqO+data	line
	Owned Repl	ReqWB	line

2.6.3 Forwarded External Requests

1. FwdReqS. Forwarded to owner in case of a ReqS.
2. FwdReqV. Forwarded to owner in case of a ReqV.
3. FwdReqO. Forwarded to owner in case of a ReqO.
4. FwdReqOdata. Forwarded to owner in case of a ReqOdata.
5. FwdInv. Forwarded to potential sharer in case of an LLC-initiated downgrade from S.
6. FwdWBAck. Forwarded to downgrading owner to signal success of downgrade. (This message is moved from the response plane to the Forward plane to help even network loads across planes.)
7. FwdRvkO. Forwarded to owner in case of an LLC-initiated downgrade from O.

2.6.4 Responses

1. RspS. Signals successful transition to S with data.
2. RspV. Data read by ReqV.
3. RspO. Signals successful transition to O.

4. RspOdata. Signals successful transition to O with data.

5. RspInvAck. Signals receipt of LLC-initiated downgrade from S.

6. RspNack. Signals failed forward request due to wrong owner.

7. RspRvkO. Signals successful LLC-initiated downgrade from O.

8. RspWT. Signals successful write-through.

9. RspWTdata. Signals successful write-through with original data.

2.6.5 Fine-grained Coherence Specialization

Fine-grained Coherence Specialization [Alsop et al., 2021] is proposed in addition to Spandex. While Spandex enables device-granularity coherence specialization, FCS takes a step further to support fine-grained coherence specialization at the granularity of each request and address. Also, FCS adds new message types such as write-through forwarding and owner-prediction that enable cache to cache coherence messages between private caches. Programs that obey a certain producer-consumer pattern can benefit from the direct communications that save round-trips to the LLC, and owning cores can further exploit temporal locality because ownership is not transferred away. Consider the example in figure 2.4, which illustrates an architecture involving one CPU and one accelerator. In traditional MESI protocol, when data is to be shared between the CPU and accelerator, ownership transfer would have to be routed through the LLC, and the read misses on the shared data will cause a downgrade process on the previous owner, thereby increasing read latency. However, with Spandex FCS, the accelerator can directly use owner prediction to read data from the CPU's private cache, and write data back to the CPU's private cache. Throughout the execution, CPU maintains ownership, and LLC is not involved in any of the transactions. FCS requires the following extensions to the existing message types:

1. ReqWTfwd. Request LLC to forward the write-through to the owning cache with data.

2. FwdWTfwd. Send write-through to the predicted owning cache directly.

3. Owner-prediction. Directly send external requests to the predicted owner.

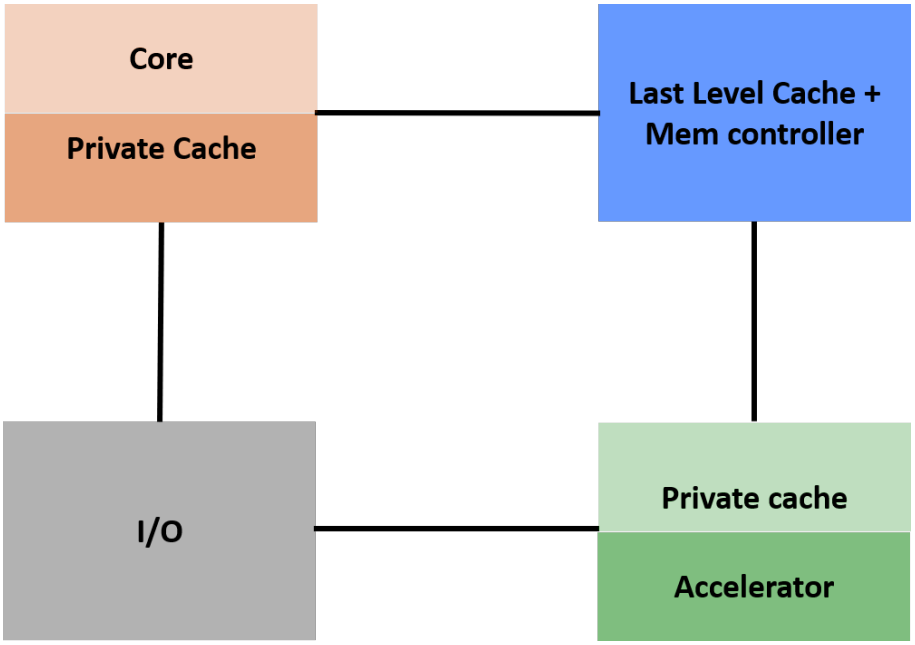

Figure 2.4: FCS Example

Chapter 3

Spandex RTL Design and Implementation

The design and implementation effort of Spandex RTL cache is divided into the following aspects:

1. Required infrastructure changes in ESP.
2. Required additional output signals in the processor core.
3. The actual cache protocol development.

This chapter describes the design and development effort in each of these aspects in detail.

3.1 NoC Re-design

In ESP, every network packet is composed of one header packet and a number of data packets. The header contains important information such as source and destination tile locations and message types. The width of each network flit used to carry the header is equivalent to the CPU address bus width plus two bits. These two bits indicate what type of flit it is, either header, body, tail, or a single flit. For example, when using RISC-V Ariane, which is a 64-bit architecture, each NoC flit has 66 bits. The original ESP NoC header is shown below.

Listing 3.1: ESP NoC Header Definition

```
// Header fields
// Let W be the global constant ARCH_BITS
|W+1         W|W-1      W-5|W-6     W-10|W-11    W-15|W-16    W-20|
|  PREAMBLE  |   Src Y   |   Src X   |   Dst Y   |   Dst X   |
|W-21      W-23|W-24       W-27|W-28        5|4    0|
|  Msg. type  |   Reserved   |  [Unused]  |LEWSN|
```

ESP message type field is only 3 bits. However, on the Request NoC plane with the most messages, Spandex has more types of messages than what is supported. Thus, the existing 3-bit message type field is not enough. Also, Spandex tracks ownership at word-granularity as opposed to MESI's line-granularity ownership. This requires every request to be accompanied by a word mask indicating which words within the current line are relevant to this specific

network packet. Since each ESP cache line is 128 bits and the smallest architectural width supported on ESP is 32 bits (the 32-bit LEON3 IP from Cobham Gaisler [Cobham Gaisler, 2020]), the largest possible word mask length is 4. As shown in listing 3.1, when running a 32-bit architecture, the width of the unused field becomes zero. Certainly, we will have more than enough bits to exploit when 64-bit cores are being used because this will immediately make the unused field width increase to 32. But doing this will leave the 32-bit architectures unsupported, violating the flexibility methodology of Spandex. Luckily, there are some workarounds. After careful study, we understand that even though the ESP source and destination coordination indices are five bits each, only 3 bits are used because the maximum mesh size supported in ESP, at the time of this project, is 8 by 8, a total of 64 tiles. This leaves each X-Y coordinate only 3-bit wide. Since there are 4 coordinates in each header (X and Y for both source and destination), we can save 8 bits in total by shrinking each index from 5 to 3. These 8 bits are enough to compensate for the increased demand of Spandex. 4 bits are used for the word mask, and 5 bits (including the original 3 bits for ESP message types) are used for the message type which is enough to encode all Spandex and FCS requests and the original ESP request types. The updated Spandex NoC header fields are shown below in listing 3.2.

Listing 3.2: Spandex NoC Header Definition

```
// Header fields
// Let W be the global constant ARCH_BITS
|W+1         W|W-1      W-3|W-4      W-6|W-7      W-9|W-10    W-12|
|  PREAMBLE  |   Src Y   |   Src X   |   Dst Y   |   Dst X   |
|W-13      W-17|W-18      w-21|w-22      W-25|W-26        5|4    0|
|  Msg. type  |  Word Mask  |  Reserved   |  [Unused]  |LEWSN|
```

Another important change to the ESP NoC architecture is the Spandex FCS cache to cache communication. Originally in ESP, there is no way for private caches to initiate a communication with another private cache. For an L2 cache, the Request plane port is only outgoing, and the Forward plane is only incoming. The only way of initiating communication is by a request to the LLC, which then forwards the message to the appropriate owners of the requested data. Spandex allows a private cache to send a request directly to a potential owner by using owner prediction. This requires the NoC to support initiating direct communication between the L2 private caches. We achieved this by adding a finite-state machine (FSM) to the Forward plane that handles sending a message directly from a private L2 cache to another.

3.2 Spandex RISC-V ISA Extension

To allow software and compilers to dynamically choose different types of load and store requests in Spandex, a set of Spandex-specific RISC-V ISA extensions is tailored for programs that may need to benefit from the efficient coherence support that Spandex provides. Any load or store that does not use the instructions in the Spandex extension of RISC-V will default to Spandex request type ReqS for load, and ReqOdata for store. For loads and stores that use the Spandex instructions, software and compilers have the flexibility to choose what type of requests to use, as indicated in table 3.2. These instructions also provide an opportunity for software to specify whether owner prediction is being used in this load or store, and the predicted owner's cache ID. As shown in table 3.1 (reprint of table 24.1 from the RISC-V official ISA specification version 20191213 [RISC-V, 2019a]):

Table 3.1: RISC-V Base Opcode Map

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Spandex leverages the two of the four available custom opcodes 0001011 and 0101011 to provide another set of FCS loads and stores available for use. The detailed ISA specification is shown in table 3.2.

Table 3.2: RISC-V Spandex ISA Extension

	31	**30-29**	**28-25**	**24**	**23-20**	**19-15**	**14-12**	**11-7**	**6-0**
SP_LD	FP	SP_TYPE	imm[7:4]	0	imm[3:0]	RS1	FUNC[2:0]	RD	0001011
SP_LD_O	FP	SP_TYPE	CID[3:0]	1	imm[3:0]	RS1	FUNC[2:0]	RD	0001011
SP_ST	FP	SP_TYPE	imm[7:4]	0	imm[3:0]	RS1	FUNC[2:0]	RD	0101011
SP_ST_O	FP	SP_TYPE	CID[3:0]	1	imm[3:0]	RS1	FUNC[2:0]	RD	0101011

These instructions specify how the Spandex request type and ownership prediction information should be encoded in each Spandex customized instruction, as shown in table 3.3. In these instructions, the FUNC[2:0] field defines the width of the transfer, which is defined the same as in the RISC-V ISA specification. Although few of the request types are yet to be implemented, we are working on completing the full FCS request set as soon as possible.

Table 3.3: Spandex Request Types

SP_TYPE	**SP_LD**	**SP_ST**
00	ReqV	ReqWT
01	ReqOdata	ReqWTfwd
10	-	ReqO
11	-	-

3.3 Supporting the RVWMO Memory Model

RVWMO stands for RISC-V Weak Memory Ordering. In short, the RVWMO memory model is a weak memory model that uses explicit memory ordering instructions to order memory operations, with some additional axioms specified by RISC-V [RISC-V, 2019a]. The supported explicit instructions are FENCE, FENCE.I, and atomic memory operations (AMO) with acquire (AQ) / release (RL) annotations. Since FENCE.I only orders operations on the same RISC-V hart, no operations are needed from Spandex because the processor pipeline is responsible for handling the ordering constraints.

A FENCE instruction can be used in RISC-V to enforce global ordering of its predecessor and successors, which are groups of instructions before and after the fence. The predecessor and successor can be either read or write (or IOs), hence there are different variants of fences as shown in table 3.4 to be considered [RISC-V, 2019a]. Particularly, FENCE.TSO is the same as a FENCE WR,WR but excluding write-to-read ordering. Thus, Spandex needs to treat FENCE.TSO the same way as FENCE WR,WR.

Two characteristics of Spandex are critical to memory ordering and need special care on these memory ordering instructions: Valid state and write-buffer. Spandex specifies that all Valid states be self-invalidated at acquire synchronization points to prevent future hits on potentially stale data. From the RVWMO memory model point of view, such self-invalidation needs to be performed at synchronizations in which global memory ordering constraints are placed on memory reads as successors. Also, the Spandex write-buffer needs to be completely drained at synchronizations in which global memory ordering constraints are placed on memory writes as predecessors. Since all instructions are committed in-order with RISC-V Ariane and all loads are blocking in the current ESP, we define the coherence actions required for Spandex when encountering different events, as shown in table 3.4.

Table 3.4: Spandex Coherence Actions on Explicit Synchronizations

Synchronization Events	Self-Invalidate	Drain Write-buffer
FENCE RW,RW	✓	✓
FENCE.TSO	✓	✓
FENCE RW,W		✓
FENCE R,RW	✓	
FENCE R,R	✓	
FENCE W,W		✓
AMO		
AMO.AQ	✓	
AMO.RL		✓
AMO.AQRL	✓	✓

3.4 Hardware Implementation

3.4.1 AXI Bus

The RISC-V Ariane processor IP communicates with the cache hierarchy or memory system using an AXI interface. The AXI interface [ARM, 2017] is a standard interconnect interface designed by ARM. It allows user-defined fields in each transaction, which makes it easy for implementing Spandex FCS ISA-cache interface. The AXI_USER field is used by Spandex to convey Spandex optimization information for loads and stores to the private cache interface which will be explained in detail later.

3.4.2 SystemC and High-Level Synthesis

The development language chosen for this project is SystemC with HLS. Designing a pure RTL cache is not an easy task because there are so many components to keep track of in hardware. Without a functional and timing-accurate model to refer to, it is difficult and time-consuming to develop Spandex from the ground up. HLS, on the other hand, provides an abstraction for hardware, allowing developers to build hardware components in high-level behavioral languages. During synthesis, HLS tools will automatically schedule and bind the hardware logic onto available resources within the given clock period target, freeing the developer from worrying about timing closure. Thus, HLS is ideal for the early stages of a hardware development project. In industry, the actual RTL design and implementation process usually follows a high-level functional model bring-up. The functional model is written in

higher level programming languages that are easy to understand and maintain. Then following the functional model, RTL structures are created and verified step by step. The Spandex cache system also follows this design methodology. With limited time and effort (only one student at the beginning of this project and throughout the first half of this project), implementing the cache in high-level languages such as SystemC is a reasonable first step to accomplish. With HLS, the model written in SystemC can be automatically converted into RTL, synthesized and implemented on the FPGA to allow full testing and debugging. Later when the functionality has been verified with the SystemC model, it is much easier to convert the SystemC model into an RTL model by hand, to achieve the most optimized hardware resource utilization.

This project only focuses on implementing the SystemC hardware model for Spandex, and allows for very easy transformation into an RTL model in the future with minimal effort. The ESP cache hierarchy is also designed in the same HLS to RTL workflow.

3.4.3 Write-Buffer Implementation in Spandex L2 Caches

Spandex supports word-granularity ownership requests (ReqO) that are part of the DeNovo protocol. However, each ESP NoC transaction packet is capable of handling requests up to all words within the entire line. If for all word-granularity requests we send out a corresponding ReqO packet only containing the requested word, it would be a huge waste of NoC router efficiency. Having too many unnecessary packets flying on the NoC not only causes network traffic congestion that delays overall system latency, but also inflates system energy consumption. For programs with high spatial locality in writes, the ESP MESI cache does not need to worry about such problems because the ESP MESI cache handles ownership at line-granularity. However, if the CPU store requests use DeNovo style ReqO, every CPU store miss will trigger a single-word ReqO to the LLC, resulting in multiple request and response packets being injected into the NoC for a single cache line. A similar issue arises with the GPU coherence protocol as well where all writes are written through to the next shared level of the memory hierarchy.

The solution to this problem is to use a write-coalescing buffer. The job of the write-buffer is to accumulate single-word write misses within a cache line, and when an eviction is needed or a release synchronization occurs, it sends out an ownership request for as many words in a line as possible to fully utilize the effort of the request and its response. Thus, when programs with high spatial locality for stores execute, Spandex does not get penalized for sending multiple single-word requests for ownership in the same line.

3.4.4 Acquire/Release Implementation in Spandex Private Caches

The Spandex cache coherence protocol assumes the running software to be data-race-free (DRF) compliant [Sinclair et al., 2015]. At phase boundaries where synchronizations are performed, read lines are self-invalidated to prevent

future read hit on stale data, and written lines are evicted from the write-buffer to make sure that the previous writes are visible to all future consumers. In the RISC-V ISA, each AMO instruction has a pair of AQ and RL annotation bits to implement ordering constraints related to the AMO. AQ marks an acquire operation that prevents any following memory accesses to be reordered before the corresponding AMO. RL marks a release operation that prevents any previous memory accesses from being reordered after the corresponding AMO. We use the AQ and RL bits associated with the AMO to perform coherence actions in the Spandex cache. If the AQ bit is set, the acquire operation requires the Spandex cache to perform self-invalidation so that all memory reads following the acquire operation will not hit on stale data from the previous phases. If the RL bit is set, the release operation requires the Spandex cache to perform a write-buffer flush so that all invisible writes (from other core and LLC's point of view) that happened before the release can be properly registered at the LLC. If both bits are set, we perform self-invalidation and write-buffer flush at the same time.

To achieve the proper self-invalidation and write-buffer drain operations in the Spandex cache, the decoder in the RISC-V Ariane core is modified to detect the AQ and RL bits in an AMO. The user-defined field within the processor's Advanced eXtensible Interface (AXI) bus [ARM, 2017] is also expanded to pass the AQ/RL information to the Spandex L2 interface.

3.4.5 Efficient Invalidation

This section discusses how to perform self-invalidations efficiently without adding excessive overhead and complexity.

In today's caches, SRAM is a major component used in storing a large amount of data such as tag, state, and data arrays. SRAMs have high storage density, thanks to the fact that many cells share the same address and bit lanes. However, to access the data within an SRAM, one needs to drive the address bus and wait for the response, which typically incurs some delay. Figure 3.1 shows a simple model of a block of SRAM which includes a single port used for both read and write.

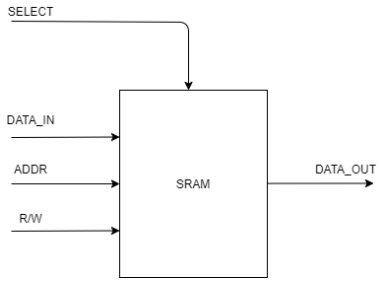

Figure 3.1: Simple Single-ported SRAM

Another form of on-chip storage, registers (flip-flops), are fast but costly. Different from SRAMs where storage

cells share both address and bit select signals, registers require layers of multiplexers to address the storage cells, and multiplexers usually occupy a lot of logic. Thus, registers are an ideal component only to store small data that requires fast access, such as states in an FSM.

The high storage density of SRAM makes it an ideal component to store a large amount of data. This is part of the main reason why in caches today we use SRAMs to store large data such as states, tags and cache lines. However, SRAMs have a significant drawback: low parallelization. Registers have a very high level of parallelization. For example, with a block of registers, one can read and write an arbitrary number of bits in one clock cycle - as long as they pay the area cost incurred by many multiplexers, and the design timing closes (the longest-path combinational logic delay plus register setup and hold times fall under the clock period). On the other hand, parallel access to a block of SRAM is limited to the number of its read and write ports. In a block of N by M SRAM (N addresses, M-bit words), one can only read and write a single M-bit word at a time, assuming the SRAM only has one R/W port. A workaround to achieve more parallelization is that one can duplicate SRAMs so that during each read one can drive different addresses on each duplicated SRAM block, to achieve parallel reads. Figure 3.2 shows an example of this method. At any given time these two SRAM blocks have the exact same copy of data which is updated simultaneously during a write. When addressed differently, these two SRAM modules can achieve parallel read accesses on two different addresses. However, even with this approach, the limitation of writing to only one single SRAM address at a time still is not lifted. In a modern system that implements its cache state array in SRAMs, whenever a whole cache flush is needed, the state SRAM needs to be invalidated address-by-address since there is no way to write to all addresses in parallel. This address-by-address invalidation manner is referred to as an “SRAM sweep” later. Thus, the latency penalty is extremely high, in the order O(N) where N stands for the number of sets or the number of cache lines.

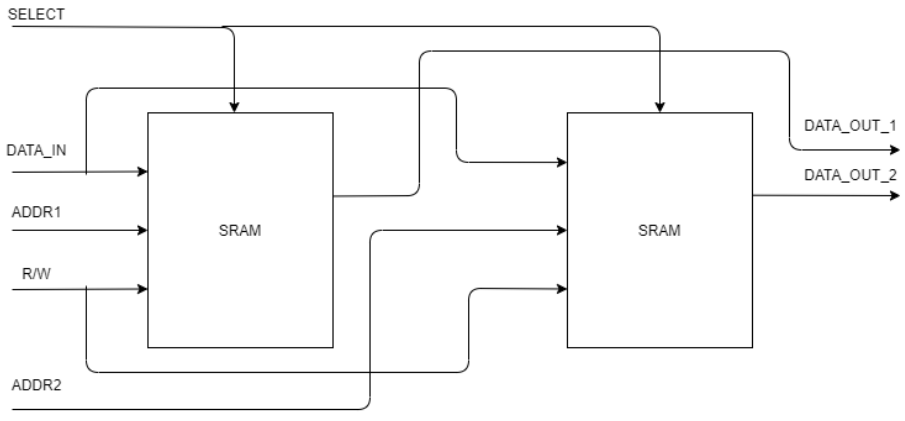

Figure 3.2: Parallel-Read SRAM Model

This problem is acceptable in normal applications running on MESI-style cache coherence systems because it is

rare that the MESI-style system needs a whole cache flush, which requires an SRAM sweep. It is true that a Spandex system does not flush its cache often either, but self-invalidations, which are quite often in some applications running on Spandex, incur SRAM sweep penalties just like a whole cache flush. During self-invalidation, all cache line states must be read out of the SRAM and then written back with a new value calculated based on its old value (Invalid if previously Valid, and unchanged if previously Registered or Shared). Spandex needs to handle this problem carefully because such self-invalidations could be very frequent in a heterogeneous system.

One intuitive solution is to store all states in registers. However, our SystemC experiments show that this solution is impractical. With a simple 32-set-2-way Spandex cache, there are so many resources consumed that Cadence® Stratus™ HLS tool fails to generate the RTL file for the GF12 technology within a reasonable amount of time, whereas an SRAM implementation can be successfully turned into RTL only occupying 0.948 mm^2 of area.

Here we propose a new solution. The key idea of this solution is that the only goal we want to achieve during a self-invalidation is to differentiate stale data with up-to-date data that will come into the cache later. During self-invalidation, the naive way is to write Invalid to every Valid state in the SRAM. However, this process is unnecessary. Valid states are only valid under the current criterion, which is "state is equal to Valid". In order to revoke its validity, there is no need to change the states themselves. Instead, we can use a new criterion that makes the old expire, for example, "Valid no longer means anything, but only Valid2 represents fresh data". In this valid state expiry technique, no writing to the state SRAM array is necessary. In the new criterion, "Valid" no longer stands for valid data, whereas "Valid2" does. By using a new value for the Valid state, we are not introducing a new state to the protocol. In fact, it is just a new rotating representation of the Valid state. The criterion shifting can be easily done by a flag modification, which is as easy as a simple counter increment in hardware. In the end, when we run out of counter values, an SRAM sweep is required. After the required SRAM sweep is completed, the flag can simply be reset to its original value. Even though SRAM sweeps are still present, we can easily reduce the number of required SRAM sweeps by multiple times, depending on the available number of stable states that can be used as such flags. The following algorithm describes in detail how this method works.

Listing 3.3: Efficient Cache Invalidation Algorithm in Hardware

```
// Data structures
constant MAX_STATE_BITS
enum StableState<bit_vector<MAX_STATE_BITS>> = {INVALID, VALID_1,
    VALID_2, ... , VALID_N, SHARED, OWNED}
StableState valid_flag = VALID_1

// main interface for self-invalidation
function Self_Invalidate
    if valid_flag < VALID_N then
        valid_flag = valid_flag + 1
    else then
        // perform cache sweep
        for each cache line
            if state != OWNED or SHARED then
                state = INVALID
            end if
        end for
    end if
end function
```

We start with a blank cache, and a set of state bits, I, $V_1 - V_n$. We put any state other than I or V from the highest state number, such as Owned. For example, if we use 3-bit state words, our Owned state would be $3b'111$. We also have a Valid flag whose default value is 1. This is the pointer to the current Valid flag, which is V_1. At any given point, the value of the Valid flag always stands for Valid. Anything less than the Valid flag is invalid, or in another word, expired. During program execution we bring some lines into the cache in V_1, S and O states. Now we reach a synchronization that requires self-invalidation. Instead of sweeping the state SRAM, we increment the Valid flag by 1, expiring V_1 and promoting V_2 as the current state indicating Valid. Now, V_1 becomes Invalid and V_2 becomes Valid, and all of this happens instantly. The previous step repeats several times, and at any given point in the cache, all V_i states are treated as Invalid if i is less than n. At some point in the future, we will have enough synchronization such that our Valid flag reaches its maximum, V_n. If we have a self-invalidation again, since there is nowhere to advance our Valid flag, an SRAM sweep is inevitable. We perform the sweep, changing everything from V_1 to V_n to I (since they are no longer valid), and retreat our Valid flag to 1. Now the whole process starts over.

For the simplest Spandex cache that only has three states, Valid, Owned and Invalid (which is essentially a DeNovo

cache), originally 2 bits are required to represent a state. This leaves us with 1 additional Valid flag position, which can reduce our total synchronization penalty by 2X, without incurring any more resource utilization to the baseline DeNovo. The more bits used, the more Valid flag positions available. If the total available Valid flags are $V_1 - V_n$, we can save the total penalty by N times. Simply expanding 1-bit width to the baseline DeNovo state array can save 6X on total penalty because there are a total of 6 Valid flag positions.

Although not yet implemented in the Spandex hardware, future improvements are still possible, which even hides the latency penalty when reverting V_n to V_1 by performing the SRAM sweep in the background. Thus, all the self-invalidation latency can be hidden, except only in rare cases when we have a quick burst of acquire synchronizations in a short amount of time.

3.4.6 MESI Translation Unit Implementation

The Spandex methodology promotes ease of development. Since Spandex is a flexible interface, an SoC designer can plug in any processor IP core or accelerator IPs on the Spandex network with minimal changes to its interface. With this design methodology in mind, we implemented a translation unit for the MESI cache that ESP provides. The purpose of the translation units (TU) is to translate original ESP coherence messages into Spandex messages, such that the MESI cache can talk with the rest of the Spandex system. As mentioned in the Spandex paper [Alsop et al., 2018], the MESI translation unit is responsible for handling word-granularity responses and external requests (forwarded requests). Since MESI uses line-granularity for all cache lines in Shared and Modified state, when there are word-granularity responses received, the translation unit must coalesce these responses until each requested word in a cache line has a valid response. Then the translation unit can remove the outstanding request from the transient state buffer. Also, the translation unit must also be able to handle word-granularity external forwarded requests to part of the line. In this case, MESI translation unit is required to handle the requested words and make sure that if there are any owned words left, they are properly written back to the LLC.

Table 3.5 describes the translated messages between Spandex and MESI L2 in ESP.

Apart from the simple message translation and partial-line response/forward handling, there are still some corner cases that need to be handled in order to integrate a Spandex translation unit into the ESP L2 MESI cache. As shown in table 3.5, in ESP, every downgrade from the Shared state in MESI results in a PutS message sent to the LLC in order to remove the downgrading cache from the LLC's internal sharer list for that cache line. However, such downgrade messages do not exist in Spandex. A Spandex L2 cache can silently downgrade from its Shared state without notifying the LLC, and the LLC, instead of keeping track of all current sharers, keeps a superset of the current sharers in which some of the caches may have already silently downgraded. In conclusion, the translation unit must immediately issue a "fake" invalidation acknowledgment to the L2 cache as soon as it detects the downgrade request, achieving the silent

Table 3.5: Message Translation Between Spandex and MESI

Spandex responses from NoC	ESP
RspS	Rsp_Data
RspO+data	Rsp_Data

Spandex forwards from NoC	ESP
Fwd_Req_S	Fwd_Get_S
Fwd_Req_O(data)	Fwd_Get_M
Fwd_Inv/Fwd_Rvk_O	Fwd_Inv
Fwd_WB_Ack	Fwd_Putack
Fwd_Req_V	(Special)

ESP responses to NoC	Spandex
Rsp_Data	Depends on the original Spandex message received
Rsp_Edata	-
Rsp_Invack	Rsp_Invack/Rsp_rvk_o

ESP requests to NoC	Spandex
GetS	ReqS
GetM	ReqO+data
PutS	-
PutM	ReqWB

downgrade requirement on Spandex. Another problem that arises from this implementation is a deadlock scenario as described below.

In order to understand this deadlock possibility that Spandex is facing, one needs to understand a special edge case in ESP. In ESP, a forwarded invalidation (FwdINV) message from the LLC to the L2 cache means an external invalidation request. During a transient state from Invalid to Shared (IS state), the ESP L2 cache blocks any FwdINV messages from being serviced. This is because in IS state, we know that there is an outstanding request for Shared state (ReqS or GetS in ESP). At the time we see a FwdINV, since ESP does not use silent eviction, we know that the ReqS has been serviced in the LLC and resulted in a transition to Shared state (S), and the response (RspS) is still on the way back. Receiving the FwdINV means that immediately after transitioning to S, LLC decides to invalidate the exact same line from all sharers (which might be a result of a racing write or LLC eviction). However, although the RspS is sent before the FwdINV, there is no ordering guarantee for the arrival time of these two packets because they are on two different network planes, the Response plane and the Forward plane. Since the distances for the two packets to travel do not differ very much (LLC-requester), the possibility of the packets arriving out of issuing order is very high. Thus, ESP L2 cache stalls servicing the forward port upon seeing that a FwdINV packet hits on the IS state. This stall condition is relieved when the RspS finally comes back to resolve the transient state IS. At this time, the FwdINV can proceed as normal.

After understanding the above ESP edge case, the potential deadlock on Spandex is much clearer to understand. Like in ESP MESI cache, a MESI cache with Spandex TU can also receive a FwdINV on IS state. However, since Spandex uses silent eviction, there is no guarantee that our previously issued ReqS has reached the LLC because the received FwdINV could be a result of the LLC trying to invalidate the superset of sharers which includes the requesting

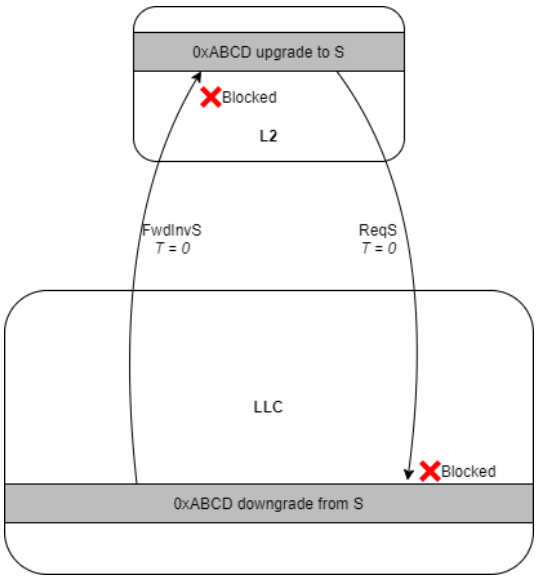

Figure 3.3: Spandex Potential Deadlock

L2 cache even though it has already downgraded from S to I. On the LLC side, if the ReqS arrives at the LLC after the LLC sends the FwdINV, the LLC will also stall the ReqS until all the pending FwdINV get a response. Consequently, if the MESI L2 with TU stalls when seeing FwdINV on IS just like ESP, a deadlock could occur. In this case, L2 waits on a RspS from the LLC, while the LLC waits on an invalidation acknowledgment. Figure 3.3 describes this deadlock scenario. Unlike with ESP, with Spandex it is currently unsafe for the L2 to expect an inflight RspS to resolve this lock. This deadlock will not be resolved without some special help from the Translation Unit.

The solution that we adopted is the following. Upon receiving a FwdINV on IS, Spandex L2 MESI TU immediately sends out an acknowledgment, but also maintains bookkeeping that such an invalidation has been received during the transition from Invalid to Shared. The acknowledgment sent by the TU should be enough to loosen the deadlock on the LLC side, allowing the LLC to resolve the pending FwdINV and proceed with servicing the blocked ReqS. After the ReqS is serviced, a RspS is guaranteed to arrive at the requesting L2 cache, thus resolving its transient state. However, when resolving the transient state IS, the TU also needs to check whether any FwdINV has been received during the transition. If so, the TU should immediately change the state of the cache line to Invalid after the processor's read request has been served. This maintains cache coherence and prevents future cache read hits on stale data because from the LLC's point of view, the MESI cache is no longer a sharer after receiving the invalidation acknowledgment, and some other processors could have already taken ownership.

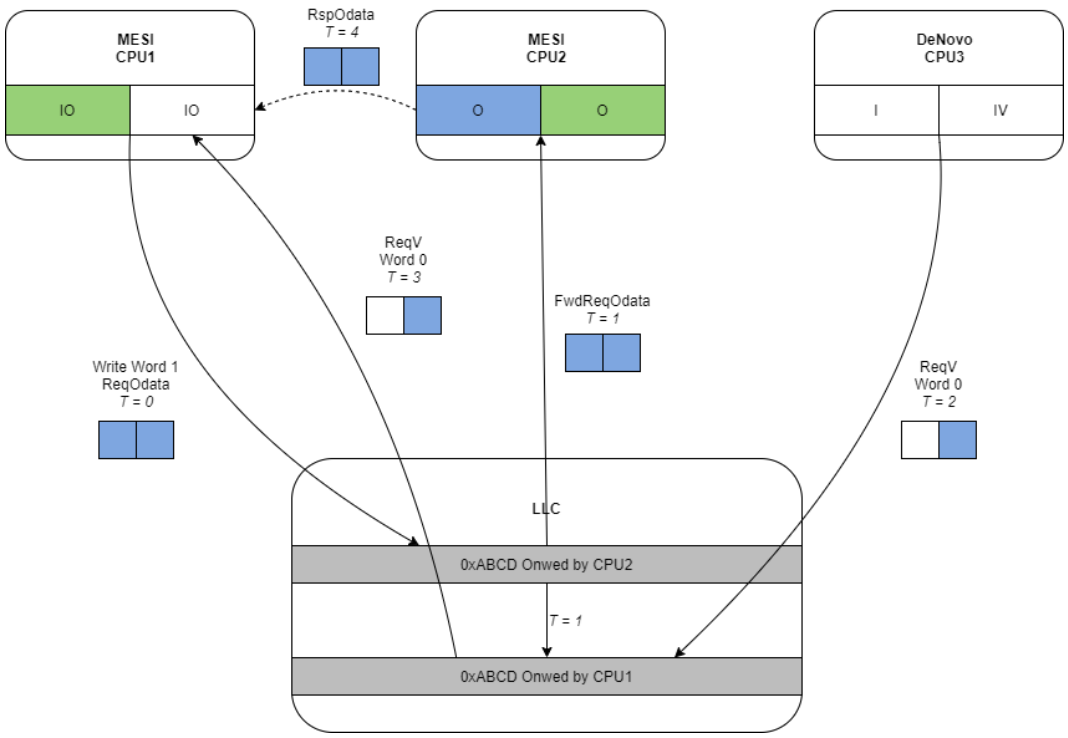

Figure 3.4: Spandex ReqV Starvation

3.4.7 DeNovo Implementation with Translation Unit

DeNovo TU implementation is fairly simple compared to MESI TU. As stated in the Spandex protocol documentation [Alsop et al., 2018], since DeNovo already handles word-granularity external requests, the only required functionality of the DeNovo TU is to handle ReqV starvation. In Spandex, ReqV is a special kind of request because it does not affect the coherence state of the LLC and does not enforce any global ordering. For this reason, when a ReqV is forwarded to an assumed owner, there is no guarantee that the owner has the most up-to-date date for the requested word, because of false sharing in MESI. Figure 3.4 visually represents this situation.

In this configuration, there are two MESI writers and one DeNovo reader. The two writers compete with each other for ownership of different words in the same cache line. In figure 3.4, green indicates the most up-to-date words.

1. At T = 0, CPU 1 is in I state, and upon CPU 1 writing Word 1, the MESI cache on CPU 1 sends out a ReqOdata message to the LLC with the word mask for both words. Simultaneously, the current owner of the cache line is CPU 2. Now, most up-to-date words for this cache line in this system are marked green.

2. At T = 1, LLC receives the ReqOdata from CPU 1, resulting in a change of ownership and a FwdReqOdata sent to CPU 2. Now CPU 1 becomes the new owner at the LLC.

3. At T = 2, CPU 3 reads Word 0, triggering a ReqV of Word 0 to the LLC.

Table 3.6: Spandex Accelerator Configuration Register

31:16	15:12	11:10	9	8	7:4	3:2	1	0
Reserved	Write Predicted Owner	Write Type	Write Owner Prediction Enable	Spandex Write Enable	Read Predicted Owner	Read Type	Read Owner Prediction Enable	Spandex Read Enable

4. At T = 3, the LLC immediately forwards the ReqV to the current owner CPU 1. However, at the current moment, there is no hardware guarantee that the RspOdata must have arrived at CPU 1, meaning the CPU 1 does not have the necessary data required to respond to the ReqV. Thus, CPU 1 must immediately Nack the request.

5. At T = 4, the RspOdata arrives at CPU 1. A retry of ReqV by CPU 3 might be successful at this point.

If after sending out RspOdata, CPU 2 sends a ReqOdata in order to write Word 1 again, then even if CPU 3 retries the ReqV, the same scenario will happen again with CPU 2 Nacking. In reality, as long as the ownership race goes on between CPU 1 and CPU 2, it is possible for the ReqV to fail indefinitely.

The solution to ReqV starvation is to implement a counter in the MSHR of CPU 3. As long as the number of retries reaches a threshold, CPU 3 must issue a request that is guaranteed to progress, such as ReqOdata.

3.4.8 LLC Implementation

The LLC implementation follows the protocol as specified in the Spandex paper [Alsop et al., 2018]. Different from the ESP LLC, the Spandex LLC is equipped with a transient state buffer in order to service other requests when previous requests cannot be serviced immediately. For example, if a ReqOdata request triggers a pending invalidation state to all sharers in the sharer list, the LLC can save this request in the transient state buffer and move on to service other request, such as a ReqWB to a different cache line that can be serviced immediately.

3.4.9 Spandex Accelerator Setup

As mentioned previously, the programs running on RISC-V cores have the ability to tag each memory access with certain Spandex properties such as request type as well as owner prediction. However, accelerators on an ESP SoC are generally non-programmable fixed-function accelerators that are not capable of executing code. The memory accesses of these accelerators are handled by the ESP DMA engine that converts the memory reads and writes generated by the accelerator into either DMA messages or coherent messages based on how the accelerator's coherence mode is configured by the running software before the accelerator is invoked. This implies that, for Spandex, if the memory access traffic from the accelerator also needs to benefit from Spandex, there needs to be an interface provided to the software to control the behavior of accelerator memory requests.

On ESP, we designed an interface for Spandex configuration by using a 32-bit configuration register (SPANDEX_REG) on the accelerator tile that can be written by the CPU before the accelerator starts running. Table 3.6 defines Spandex property fields within the SPANDEX_REG.

Software configures a Spandex accelerator by writing a 32-bit word into the SPANDEX_REG register on the accelerator tile. The specific fields of this register are wired to the interface of the accelerator's Spandex L2 cache so that the special memory requests can be properly interpreted.

Following is an example piece of C code that configures an accelerator before its invocation.

Listing 3.4: Example Accelerator Configuration

```
#define SPANDEX_REQ_WT_FWD 1
#define SPANDEX_REQ_V      1
typedef union
{
  struct
  {
    unsigned char r_en   :  1;
    unsigned char r_op   :  1;
    unsigned char r_type :  2;
    unsigned char r_cid  :  4;
    unsigned char w_en   :  1;
    unsigned char w_op   :  1;
    unsigned char w_type :  2;
    unsigned char w_cid  :  4;
    uint16_t      rsrvd2 : 16;
  };
  uint32_t spandex_reg;
} spandex_config_t;

spandex_config_t spdx_cfg = 0;
// Enable Spandex, Owner Prediction
spdx_cfg.r_en = 1;
spdx_cfg.w_en = 1;
spdx_cfg.r_op = 1;
spdx_cfg.w_op = 1;
// Set access types to write-through forward and requ
```

```
spdx_cfg.w_type = SPANDEX_REQ_WT_FWD;
spdx_cfg.r_type = SPANDEX_REQ_V;
// Predicted owner's ID is 15
spdx_cfg.w_cid = 0xf;
spdx_cfg.r_cid = 0xf;
// send the configuration to accelerator
iowrite32(dev, SPANDEX_REG, spdx_cfg);
generate_data();
release();
// Send start signal to accelerator
iowrite32(dev, CMD_REG, CMD_MASK_START);
```

Here the first IOWRITE to SPANDEX_REG means that all consequent memory accesses issued by the accelerator will have both Spandex optimization and owner prediction turned on, while using Spandex request type ReqV and ReqWTfwd, and the predicted owner has a cache ID of 15.

Unlike processors, accelerators in ESP are not programmable so they cannot send synchronization AMOs with coherence action information for Spandex like AQ and RL. In Spandex, the synchronization of the accelerator side can be achieved by utilizing the fence channel initially designed for the CPU. Since the accelerator synchronization requires both self-invalidation and write-buffer flush to make sure that all pending write operations in the write-buffer are properly registered in the LLC, and also that all stale data is invalidated in its private cache, we can send a full R/W fence signal to its private cache when the accelerator asserts the done signal, in preparation for the next invocation.

3.4.10 Spandex and ESP Integration

Graphical Interface

In order to allow easy options for the users to try Spandex in their designs, a drop-down box is added in the ESP GUI configuration window. As shown in figure 3.5, users can mix and match private cache protocols for any compute tile. Available selections include MESI, DeNovo, GPU and Spandex. This configuration setting is per-tile, allowing the users to have a fully heterogeneous cache hierarchy with a different cache coherence protocol used in each tile, while coherence is maintained by the Spandex LLC.

Spandex Open-Source

Spandex is open-source and available at the following GitHub repositories.

1. ESP Infrastructural Support

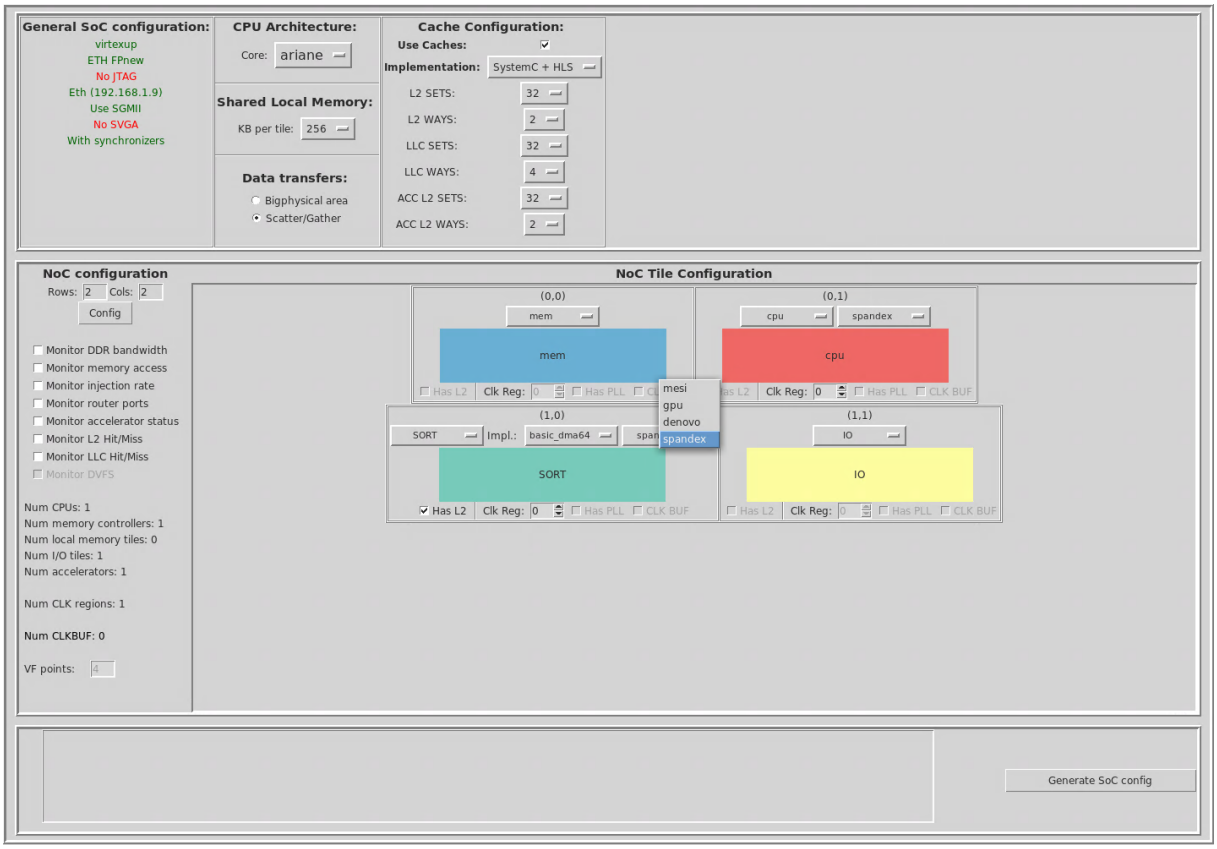

Figure 3.5: Spandex Configuration Window

2. RISC-V Ariane Core with Spandex ISA Extension

3. RISC-V Ariane Atomics Wrapper for Spandex

4. Spandex Caches Repository

Chapter 4

Evaluation

4.1 Evaluation Setup

The evaluation for the Spandex protocol in hardware is done using ESP accelerators. We choose SORT and FFT accelerators because they are good representatives of a producer-consumer relationship with the CPU.

For each of the benchmarks, we use one Ariane CPU tile (and its integrated write-through L1 cache) with a Spandex L2 cache, one ESP hardware accelerator with a Spandex private L2 (which we refer to as an L2 for uniformity), and an ESP memory tile with a Spandex LLC.

Both L2 caches are 512 sets by 2 ways, with 16 bytes per cache line. The LLC is 1024 sets by 8 ways, also with 16 bytes per line. The CPU L1 is black-boxed, and its size does not matter because the shared data only exists in the accelerator-specific memory region which is non-cacheable in the CPU's L1 (due to a temporary constraint in ESP at the time of this project). The L2 cache has a 4-entry transient-state holding register (similar to a miss-status holding register or MSHR) and a 4-entry write-buffer. The LLC has a 4-entry transient-state holding register. Each entry in either the transient-state holding register or the write-buffer can handle information for up to a cache line. The L2 hit latency in both Spandex and MESI is 4 cycles. An AMO.RL hit in the Spandex L2 takes around 80 cycles to fully complete, that is, to evict the entire write-buffer and collect all responses from the LLC. In the best scenario, an AMO.AQ hit in the Spandex L2 has the same latency as the L2 hit latency because of the instantaneous self-invalidate. As discussed before, the self-invalidation penalty will eventually be incurred after a number of iterations, but in this evaluation we do not take this penalty into account, since we are still actively working on a solution that further reduces the self-invalidate penalty, and that eventually even completely hides this latency by simultaneously handling self-invalidate and memory requests, as explained in the future work chapter. In the configuration we used for this experiment, each Spandex stable state in the private cache consumes three bits. Excluding Invalid, Shared and Owned, there are five state numbers for Valid. This means for every five self-invalidations, the penalty for cache sweep is going to be incurred only once. In this specific experiment, we run the accelerator three times so that the self-invalidation penalty will never hit, and collect the performance metrics for the second iteration, in three ways:

1. the number of CPU cycles taken for the CPU and the accelerator to run

2. the number of injected flits on each network plane by CPU, accelerator and the LLC

3. the total number of flit-hops of network traffic, which can be interpreted as the active energy consumed by the NoC

We evaluate the hardware implementation using the following three Spandex FCS settings on the Xilinx VCU118 FPGA evaluation board:

- Baseline Spandex, where all writes are ReqO, and all reads are ReqV
- Spandex FCS with write-through forwarding (WTfwd), where the accelerator uses ReqWTfwd for all its writes
- Spandex FCS with owner prediction (OP), where the accelerator directly sends all reads (ReqVo) and all writes (ReqWTfwdo) to the CPU's private cache

We compare the Spandex performance in the above three settings with ESP baseline performance running on LLC-recall coherent DMA mode and fully coherent mode.

During each evaluation, the micro-benchmark is run for multiple iterations. Within each iteration, there are two phases. During the first phase, Accelerator phase, the CPU invokes the accelerator and waits to read the done signal until the accelerator finishes. During the second phase, CPU phase, the CPU acquires the accelerator output data and begins to read and validate the data. Finally, the CPU also writes to the same buffer again and releases it in order to invoke the accelerator in the next iteration. All performance metrics are collected during one of the iterations in the steady-state, which is after CPU cold starts with the beginning iteration.

As an example, ESP provides a SORT accelerator which is designed to sort a given length of floating-point array and write its output to a designated output buffer. The micro-benchmark consists of a bare-metal C driver for the SORT accelerator that generates input and then validates the output of the sort accelerator.

We emphasize that the results presented here are the first results from the Spandex hardware implementation. They are neither an upper nor lower bound for the benefits that can be obtained from Spandex. A more thorough evaluation is part of our future work in chapter 5.

4.2 Execution Phase Breakdown Analysis

4.2.1 SORT Accelerator

During the benchmark program, the SORT accelerator runs in-place, modifying the content of a buffer that is filled with random values generated by the CPU.

First, when the program starts, we enter a startup phase where the CPU first writes an array of random floats into the buffer. The buffer contains two batches of 64 floating point numbers, a total size of 512 bytes. The buffer is not cached in CPU L1 but is cached in both L2 and LLC. After the CPU is done writing the buffer, we enter the steady-state. The CPU writes a start message into a memory-mapped control register in the sort accelerator. Then the program enters the Accelerator phase where the accelerator starts the sorting process as soon as it receives the start message. The accelerator reads data from the buffer, sorts the array, and writes the results back to the same buffer. The accelerator uses ping-pong buffering in its scratchpad to load, compute and store the two batches in a pipelined manner. This process requires the accelerator to read from the entire buffer once and write to the entire same buffer once. While the accelerator is running, the CPU keeps polling another memory-mapped status register to check if the SORT accelerator is done. The accelerator writes a "done" message into the status register to let the CPU know the completion of the Accelerator phase. After the CPU receives the done message, the program enters the CPU phase where the CPU validates the result by reading back the buffer to check if the array of floating-point numbers is in the expected ascending order, and then fills the same array with new random data to be sorted. This process requires the CPU to read the entire buffer once, and write the entire buffer once.

Cycles

Figure 4.1 shows the performance evaluation results for the SORT micro-benchmark, broken down into different phases.

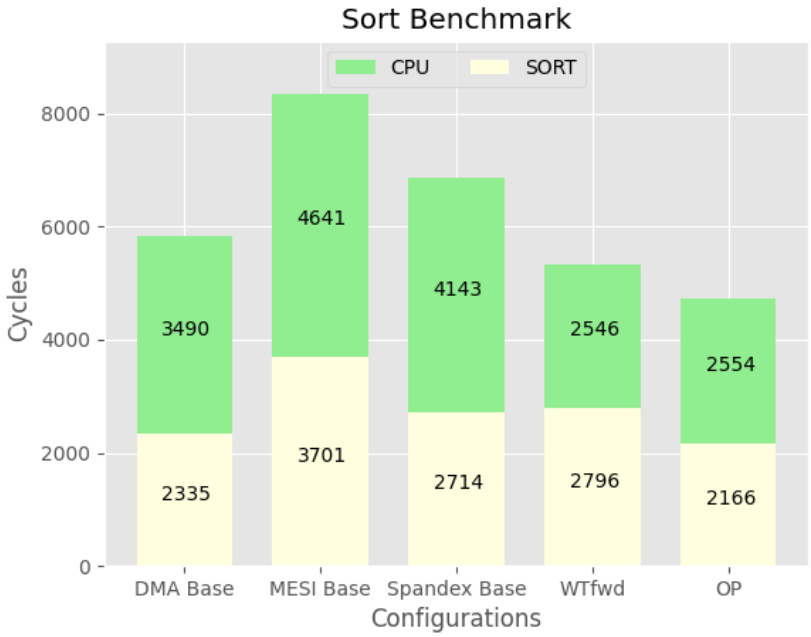

Figure 4.1: Spandex Sort Cycles

In DMA base, the accelerator sends requests directly to LLC, without using its own private cache. Since a single read request causes a burst of read responses, the DMA base configuration runs very fast in the Accelerator phase (yellow) compared to MESI baseline. It takes more time for the accelerator to read data in MESI baseline because the accelerator read channel is blocking, meaning a read for the next cache line cannot issue until the previous read response has come back.

The test results show that Spandex Base has a shorter execution time for Accelerator phase because the accelerator writes data at 64-bit word granularity, which is only half a line. In this case, Spandex Baseline which uses word-granularity ReqO (which is even further coalesced in the write-buffer) is more efficient than the MESI cache. With MESI, the first write to a line results in a ReqOdata being sent to LLC and the L2 going into a transient state (pending ownership) that blocks future writes on the same line (a design choice in ESP). Thus, MESI writes are less efficient than Spandex writes because Spandex writes are non-blocking in that no transient state is blocking the subsequent write (to the same line) to be serviced. In the CPU phase, we see that Spandex Base performs better than MESI also because of the non-blocking word-granularity writes. Although this comparison seems not fair for MESI because the baseline MESI implementation does not support writes during pending ownership, the fact that Spandex does not have to worry about this problem still shows its flexibility.

ReqWTfwd has similar Accelerator phase performance compared to Spandex Base as expected. In the CPU phase, since it uses ReqWTfwd to perform a direct update of the owned data in the CPU's private cache, the CPU benefits

from read hits since there is no ownership transfer during Sort phase.

Spandex OP is another improvement based on Spandex WTfwd. With owner-prediction enabled, the predicted owner is pre-programmed by the CPU to be the CPU's own private cache. Since the buffer size is significantly smaller than the cache size, the entire buffer is stored in the CPU's L2 cache during the sort. Thus, the owner prediction accuracy is 100%. The latency of accelerator read can be reduced because of direct cache to cache communication (Reader-Owner-Reader). Thus, we see a significant reduction in Accelerator phase. The CPU phase compared to WTfwd is similar as expected, because in both cases CPU sees hits in its private cache.

Overall, when compared to the MESI baseline implementation that ESP provides, FCS OP achieves 43.42% total execution time reduction, or 1.77X performance speedup, compared to MESI for one steady-state iteration without the self-invalidate penalty which eventually can be hidden. When compared to LLC-recall DMA, the performance speedup is 1.23X, slightly less than MESI but still a considerable amount.

Injected Flits

Table 4.1 shows the network traffic analysis of the SORT benchmark. The reduction from MESI to Spandex Baseline is because Spandex ReqO does not need to have the old data in response as MESI does when using ReqOdata. The reduction from Spandex Baseline to WTfwd is because the accelerator directly writes to the CPU's private cache, so there is no ownership transfer. The reduction from WTfwd to OP is because even LLC traffic during writes is eliminated because of the inter-private cache transfer.

The total injected network flit count is summarized in figure 4.2. Compared to MESI, OP can achieve 68.13% total network injected flits reduction, or 3.14X efficiency improvement. On the other hand, when compared to DMA, OP can achieve 1.29X efficiency improvement.

Table 4.1: SORT Network Traffic

	DMA	MESI	BASE	WTFWD	OP
CPU FLIT (REQ/FWD/RSP)	94/0/128	164/0/320	160/0/192	34/0/192	34/0/192
ACC FLIT (REQ/FWD/RSP)	-	128/0/320	128/0/192	192/0/0	0/192/0
LLC FLIT (REQ/FWD/RSP)	0/64/180	0/256/312	0/256/60	0/192/60	0/0/60
DMA FLIT (ACCREQ/LLCRSP)	84/68	-	-	-	-
TOTAL INJECTED FLIT	618	1500	988	670	478
OP FLIT REDUCTION	22.65%	68.13%	51.62%	28.66%	0.00%

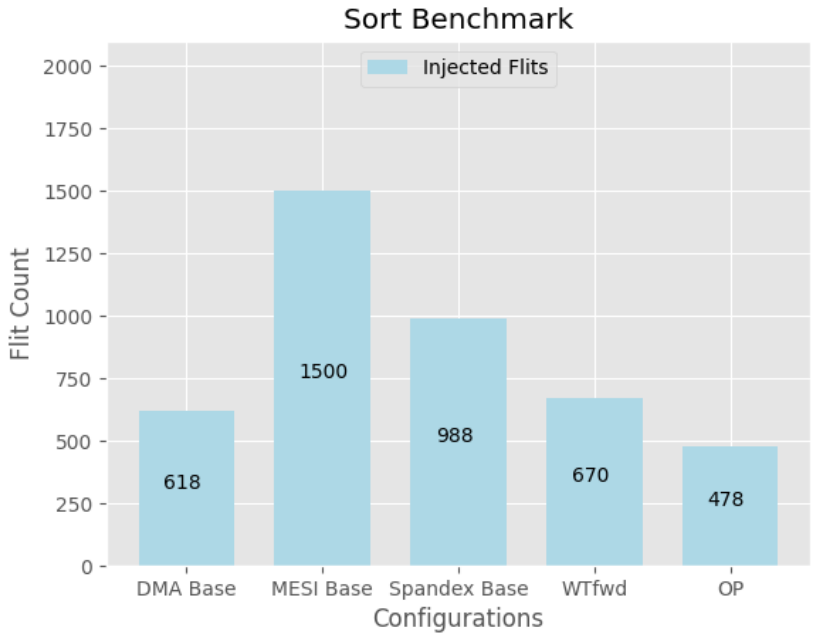

Figure 4.2: Spandex Sort Flits

Flit Hops

Based on the number of injected flits we can calculate the total flit-hop metrics in the NoC. In an 8 by 8 SoC with CPU at bottom right, SORT at bottom left, and LLC at top left corners, the distance between CPU and SORT, as well as between LLC and SORT, is 7 hops, while the distance between CPU and LLC is 14 hops. The logical distance for any flit to travel does not change on the NoC implementing static routing algorithm.

Figure 4.3 shows the total number of flit-hops in the network traffic during the specific iteration. We can see that OP is able to provide 72.81% reduction, or 3.68X efficiency improvement over MESI, and 1.90X improvement over DMA.

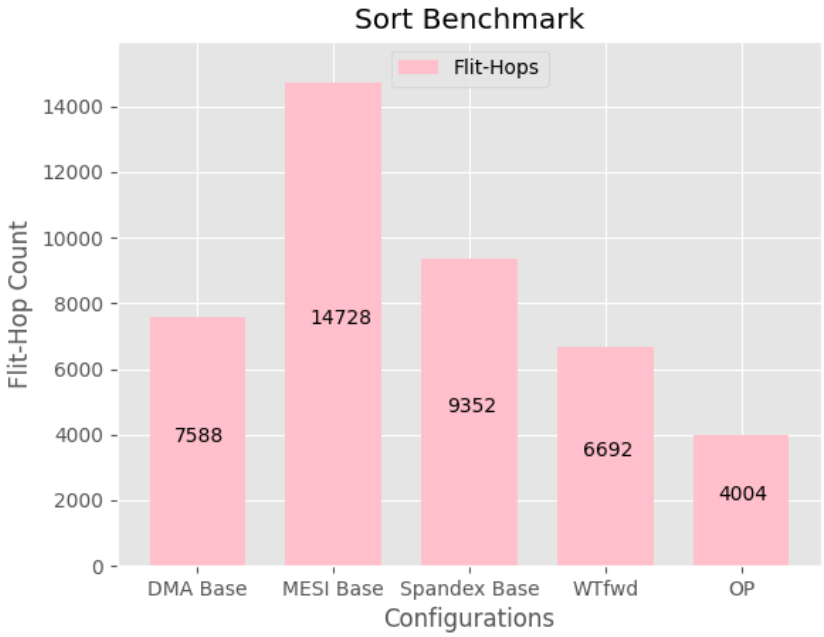

Figure 4.3: Spandex Sort Flit-Hops

4.2.2 FFT Accelerator

For completeness, we also tested our design with the FFT accelerator. FFT is another accelerator in ESP. Upon receiving its input data, it performs an FFT algorithm on the input data and stores it back. Similar to SORT, the configuration is almost the same, and the steady-state execution is also broken down into two phases, where Accelerator phase invokes the FFT device, and the CPU phase reads and writes the new input data again for the next iteration. The only difference is that with an 8 by 8 SoC we are seeing some timing violations that fail the bitstream generation, so we had to reduce the size of the SoC from 8 by 8 to 7 by 7.

Just like SORT, FFT also runs in-place. First, when the program starts, the CPU enters the cold start phase where it stores the input data of 1KB for the accelerator to work on. After generating data, the CPU enters the steady-state and invokes the accelerator. After the Accelerator phase is completed, the program enters the CPU phase and reads back the FFT output data and writes new data to be computed. Same as in SORT, we run the micro-benchmark for three iterations, and the same performance metrics are collected for the second iteration in steady-state.

Cycles

Figure 4.4 shows the performance evaluation results for the FFT micro-benchmark, broken down into different phases.

For the same reasons discussed above, in OP we are seeing a total of 37.67% execution time reduction, or 1.60X

performance speedup over MESI, and 12.42% execution time reduction, or 1.14X performance speedup over DMA.

The reason why performance speedup of Spandex over MESI/DMA in FFT is not as good as SORT is because: in each iteration, the SORT accelerator uses ping-pong buffering to compute two batches in a pipelined manner (load input of next batch while sorting the current batch). This technique hides some compute latency and causes the memory system to be stressed more than FFT.

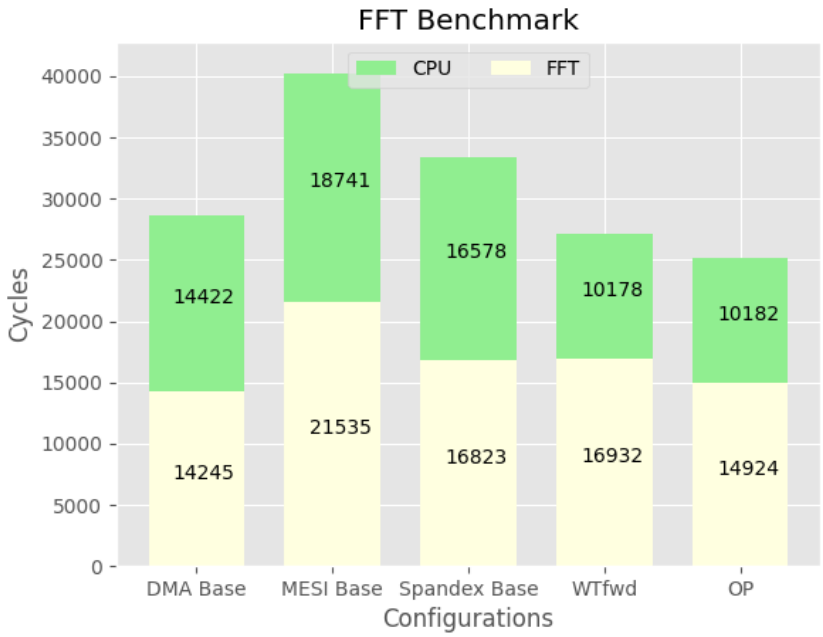

Figure 4.4: Spandex FFT Cycles

Injected Flits

Table 4.2 shows the network traffic analysis of the FFT benchmark (for three iterations). Also, the total injected network flit count is summarized in figure 4.5. Compared to MESI, OP can achieve 71.83% total network injected flits reduction, or 3.55X efficiency improvement. On the other hand, when compared to DMA, OP can achieve 1.31X efficiency improvement.

Table 4.2: FFT Network Traffic

	DMA	MESI	BASE	WTFWD	OP
CPU FLIT (REQ/FWD/RSP)	812/0/1540	1578/0/3848	1602/0/2308	66/0/2308	66/0/2310
ACC FLIT (REQ/FWD/RSP)	-	1538/0/3840	1538/0/2304	2306/0/0	2/2306/0
LLC FLIT (REQ/FWD/RSP)	0/770/1616	0/3074/3152	0/3076/112	0/2308/112	0/2/112
DMA FLIT (ACCREQ/LLCRSP)	786/773	-	-	-	-
TOTAL INJECTED FLIT	6297	17030	10940	7100	4798
OP FLIT REDUCTION	**23.80%**	**71.83%**	**56.14%**	**32.42%**	**0.00%**

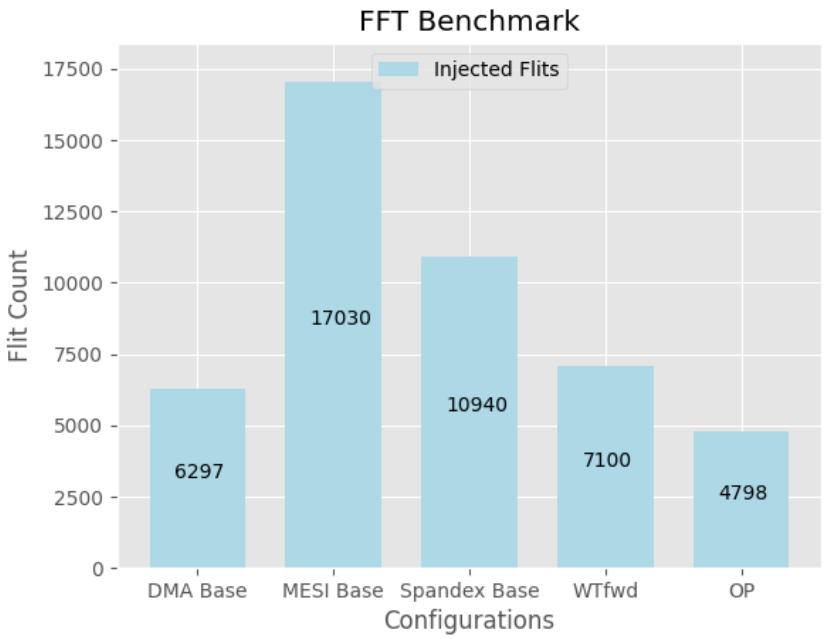

Figure 4.5: Spandex FFT Flits

Flit Hops

Based on the number of injected flits, we can calculate the total flit-hop metrics in the NoC. In an 7 by 7 SoC with CPU at bottom right, FFT at bottom left, and LLC at top left corners, the distance between CPU and FFT, as well as between LLC and FFT, is 6 hops, while the distance between CPU and LLC is 12 hops.

Figure 4.6 shows the total number of flit-hops in the network traffic during the specific iteration. We can see that OP is able to provide 81.12% reduction, or 5.30X efficiency improvement over MESI, and 2.22X improvement over DMA.

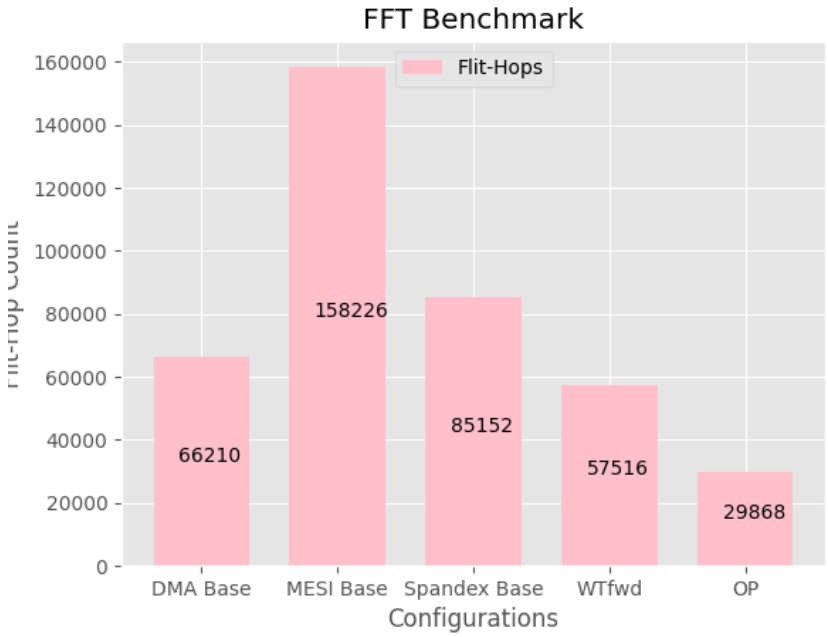

Figure 4.6: Spandex FFT Flit-Hops

Chapter 5

Conclusions and Future Work

Spandex can enable coherence specialization at device granularity, whereas Spandex FCS further extends the specialization and flexibility to address and request granularity within a device. Also, FCS provides other efficient specializations such as direct cache to cache transfer. By using Spandex and FCS, we can achieve better performance without sacrificing complexity and programmability in a heterogeneous SoC.

By implementing Spandex in hardware, we demonstrate the benefits of Spandex in a real-world FPGA-based heterogeneous system. This work strengthens the Spandex methodology by demonstrating the feasibility of implementing flexible coherence on an SoC architecture based on a NoC interconnect, while preserving as much of the existing hardware IPs and software stack as possible. During the development of this project, minimal changes were made to the existing processor IP cores, accelerator interfaces and network hierarchy. The flexibility of such development flow will encourage more hardware designers and software developers to consider Spandex as their solution for flexible and scalable on-chip cache coherence.

We would like to evaluate Spandex with micro-benchmarks where the accelerator also does not incur misses - with double buffering, the CPU can also use cache-to-cache transfer to write data directly into the accelerator cache. Further, in a system that can exploit accelerator level parallelism, such cache to cache transfer between accelerators can provide additional benefits depending on the baseline implementation. We are also considering replacing internal accelerator memories (scratchpads and buffers) with stash-like structures [Komuravelli et al., 2015] that can be part of the coherent global address space with optimizations like the cache to cache transfers described here and the attendant programmability advantages. There is much to be explored with this infrastructure.

Besides the above evaluations and forward-looking implementations, the Spandex implementation can also be improved in the following ways, in the order of decreasing importance:

1. Boot Linux SMP on multicore Ariane.

2. Strengthen the RISC-V ISA design to include several missing FCS request types such as flexible atomics execution.

3. Currently Spandex RISC-V ISA extension can be improved. Further down the path, we can to exploit oppor-

tunities such as in-cache hardware owner predictors, instead of exposing cache IDs to the software or compiler like in the current ISA. Also, in order to make room for the FCS encoding, we carved out quite a few immediate offset bits in the RISC-V instruction. A thorough evaluation of how this affects code density is needed.

4. Currently Spandex write-buffer only supports word-granularity coalescing, and sub-word granularity writes have to use ReqOdata. If extended to byte-granularity, the write-buffer can also support sub-word granularity writes more efficiently.

5. Currently the ESP NoC routers do not respect Spandex bitmasks, meaning that the NoC will still transfer a word even though the Spandex bitmask for that word is not set in the packet header. This wastes energy and increases latency when not all words are valid in a packet. Improvements can be made to allow routers to drop network flits when they see that some words in the packet are not valid.

This work can be extended to accommodate many other SoC design platforms, such as OpenPiton [Balkind et al., 2016]. One missing piece that will greatly ease wider adoption is the Spandex compiler. The FCS paper [Alsop et al., 2021] already presents a tool that determines the best request type for each load and store based on a set of heuristics. Thus, the only additional functionality required for the Spandex compiler compared to a normal GCC or LLVM compiler is to parse the coherence request types from the tool to generate the appropriate Spandex RISC-V instructions in the application binary. The Spandex compiler can provide software developers a more accessible interface for programming with high-performance and efficient coherence support.

With further extensions and development, we believe this project will benefit the accelerator design and SoC design community in the long term, and further promote the heterogeneous system and hardware-software co-design methodology for many applications like mixed reality and machine learning. In addition, there are many opportunities for collaborations with compilers and workflows which target heterogeneous systems, and many more innovative research projects can benefit from the Spandex hardware.

References

[Adve and Hill, 1990] Adve, S. V. and Hill, M. D. (1990). Weak ordering—A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 2–14, New York, NY, USA. Association for Computing Machinery.

[Alsop et al., 2021] Alsop, J., Na, W. T., Sinclair, M. D., Grayson, S., and Adve, S. V. (2021). A case for fine-grain coherence specialization in heterogeneous systems.

[Alsop et al., 2018] Alsop, J., Sinclair, M., and Adve, S. (2018). Spandex: A flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 261–274.

[ARM, 2017] ARM (2017). AMBA AXI and ACE protocol specification.

[Balkind et al., 2016] Balkind, J., McKeown, M., Fu, Y., Nguyen, T., Zhou, Y., Lavrov, A., Shahrad, M., Fuchs, A., Payne, S., Liang, X., Matl, M., and Wentzlaff, D. (2016). OpenPiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA. ACM.

[Carloni, 2016] Carloni, L. P. (2016). Invited: The case for embedded scalable platforms. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Choi et al., 2011] Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S. V., Adve, V. S., Carter, N. P., and Chou, C. (2011). Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166.

[Cobham Gaisler, 2020] Cobham Gaisler (2020). GRLIB IP Library User's Manual.

[Giri et al., 2018] Giri, D., Mantovani, P., and Carloni, L. P. (2018). NoC-based support of heterogeneous cache-coherence models for accelerators. In *Proceedings of the Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*.

[Komuravelli et al., 2015] Komuravelli, R., Sinclair, M. D., Alsop, J., Huzaifa, M., Kotsifakou, M., Srivastava, P., Adve, S. V., and Adve, V. S. (2015). Stash: Have your scratchpad and cache it too. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 707–719.

[RISC-V, 2019a] RISC-V (2019a). Volume 1, unprivileged spec v. 20191213.

[RISC-V, 2019b] RISC-V (2019b). Volume 2, privileged spec v. 20190608.

[Sinclair et al., 2015] Sinclair, M. D., Alsop, J., and Adve, S. V. (2015). Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 647–659.

[Taylor, 2013] Taylor, M. B. (2013). A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19.

[Zaruba and Benini, 2019] Zaruba, F. and Benini, L. (2019). The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640.